Synthesis of Fault-Tolerant Embedded Systems with Checkpointing and Replication

Izosimov, Viacheslav; Pop, Paul; Eles, Petru; Peng, Zebo

Published in:
International Workshop on Electronic Design, Test & Applications

Link to article, DOI:
10.1109/DELTA.2006.83

Publication date:
2006

Synthesis of Fault-Tolerant Embedded Systems with Checkpointing and Replication

Viacheslav Izosimov, Paul Pop, Petru Eles, Zebo Peng
Computer and Information Science Dept., Linköping University, Sweden
{viaiz, paupo, petel, zebpe}@ida.liu.se

Abstract

We present an approach to the synthesis of fault-tolerant hard real-time systems for safety-critical applications. We use checkpointing with rollback recovery and active replication for tolerating transient faults. Processes are statically scheduled and communications are performed using the time-triggered protocol. Our synthesis approach decides the assignment of fault-tolerance policies to processes, the optimal placement of checkpoints and the mapping of processes to processors such that transient faults are tolerated and the timing constraints of the application are satisfied. We present several synthesis algorithms which are able to find fault-tolerant implementations given a limited amount of resources. The developed algorithms are evaluated using extensive experiments, including a real-life example.

1. Introduction

Safety-critical applications have to function correctly and meet their timing constraints even in the presence of faults. Such faults can be permanent (i.e., damaged microcontrollers or communication links), *transient* (e.g., caused by electromagnetic interference), or intermittent (appear and disappear repeatedly). The transient faults are the most common, and their number is continuously increasing due to the continuously raising level of integration in semiconductors.

Researchers have proposed several hardware architecture solutions, such as MARS [17], TTA [18] and XBW [4], that rely on hardware replication to tolerate a single permanent fault in any of the components of a fault-tolerant unit. Such approaches can be used for tolerating transient faults as well, but they incur very large hardware cost, which becomes unacceptable in the context of a potentially large number of transient faults. An alternative to such purely hardware-based solutions are approaches such as checkpointing, replication and re-execution.

For the case of preemptive on-line scheduling, researchers have shown how the schedulability of an application can be guaranteed at the same time with appropriate levels of fault-tolerance [1, 2, 11]. For processes scheduled using fixed-priority preemptive scheduling, Punnekkat et al. [23] derive the optimal number of checkpoints for a given task in isolation, and show that this does not correspond to the global optima. However, in many safety-critical applications, static off-line scheduling is the preferred option for ensuring both the predictability of worst-case behavior, and high resource utilization [16].

The disadvantage of static scheduling approaches, however, is their lack of flexibility, which makes it difficult to integrate tolerance towards unpredictable fault occurrences. Thus, researchers have proposed approaches for integrating fault-tolerance into the framework of static scheduling. A simple heuristic for combining several static schedules in order to mask fault-patterns through replication is proposed in [5], without considering the timing constraints of the application. This approach is used as the basis for cost and fault-tolerance trade-offs within the Metropolis environment [20]. Graph transformations are used in [3] in order to introduce replication mechanisms into an application. Such a graph transformation approach, however, does not work for checkpointing and re-execution, which have to be considered during the construction of the static schedules.

Checkpointing in the context of static scheduling has received limited attention [8, 19]. The process model in [9] supports the application of checkpoints, while [19] has proposed a high-level synthesis algorithm for ASICs that introduces low overhead checkpoints in a static schedule. When re-execution (which can be equated to a single-checkpoint scheme) is used in a distributed system, Kandasamy [14] proposes a list-scheduling technique for building a static schedule that can mask the occurrence of faults, thus making the re-execution transparent. Slacks are inserted into the schedule in order to allow the re-execution of processes in case of faults. The faulty process is re-executed, and the processor switches to a contingency schedule that delays the processes on the corresponding processor, making use of the slack introduced. The authors propose an algorithm for reducing the necessary slack for re-execution. This algorithm has later been applied to the fault-tolerant transmission of messages on a time-division multiple-access bus (TDMA) [15].

Applying such fault-tolerance techniques introduces overheads in the schedule and thus can lead to unschedulable systems. Very few researchers [14, 19, 20] consider the optimization of implementations to reduce the overheads due to fault-tolerance and, even if optimization is considered, it is very limited and does not include the concurrent usage of several fault-tolerance techniques. Moreover, the application of fault-tolerance techniques is considered in isolation, and thus is not considered in relation to other levels of the design process, including mapping, scheduling and bus access optimization. In addition, the communication aspects are not considered or very much simplified.

In this paper, we consider hard real-time safety-critical applications mapped on distributed embedded systems. Both the processes and the messages are scheduled using non-preemptive *static cyclic scheduling*. The communication is performed using a communication environment based on *the time-triggered protocol* [16], thus the communication overheads are taken into account. We consider two distinct fault-tolerance techniques: process-level local *checkpointing* with rollback recovery (with a constant checkpointing interval) [6], which provides time-redundancy, and active *replication* [21], which provides space-redundancy. We show how checkpointing and replication can be combined in an optimized implementation that leads to a sched-

ulable fault-tolerant application without increasing the amount of employed resources.

In [13] we have proposed algorithms for the scheduling and optimization of fault-tolerant embedded systems using re-execution and replication. In this paper, we have extended this approach to take into account checkpointing. Thus, we propose a synthesis approach in the context of checkpointing and replication: the optimization of the number of checkpoints, the assignment of fault-tolerance techniques to processes, and the mapping of processes to processors such that the application is schedulable and no additional hardware resources are necessary.

2. System Architecture

2.1 Hardware Architecture and Fault Model

We consider architectures composed of a set $\mathcal{N}$ of nodes which share a broadcast communication channel. Every *node* $N_i \in \mathcal{N}$ consists, among others, of a communication controller and a CPU (see Figure 1a).

The communication controllers implement the protocol services and run independently of the node's CPU. We consider the time-triggered protocol (TTP) [16] as the communication infrastructure. However, the research presented is also valid for any other TDMA-based bus protocol that schedules the messages statically based on a schedule table like, for example, the SAFEbus [12] protocol used in the avionics industry.

The TTP has a replicated bus and each node N_i can transmit only during a predetermined time interval, the so called TDMA slot S_i, see Figure 1b. In such a slot, a node can send several messages packed in a frame. A sequence of slots corresponding to all the nodes in the TTC is called a TDMA round. A node can have only one slot in a TDMA round. Several TDMA rounds can be combined together in a cycle that is repeated periodically. The TDMA access scheme is imposed by a message descriptor list (MEDL) that is located in every TTP controller. The MEDL serves as a schedule table for the TTP controller which has to know when to send/receive a frame to/from the communication channel.

In this paper we are interested in fault-tolerance techniques for tolerating transient faults, which are the most common faults in today's embedded systems. If permanent faults occur, we consider that they are handled using a technique such as hardware replication. Note that an architecture that tolerates *n* permanent faults, will also tolerate *n* transient faults. However, we are interested in tolerating a much larger number of transient faults than permanent ones, for which using hardware replication alone is too costly.

We have generalized the fault-model from [14] that assumes that one single transient fault may occur on any of the nodes in the system during the application execution. In our model, we consider that at most *k* transient faults[1] may occur anywhere in the system during one operation cycle of the application. Thus, not only several transient faults may occur simultaneously on several processors, but also several faults may occur on the same processor.

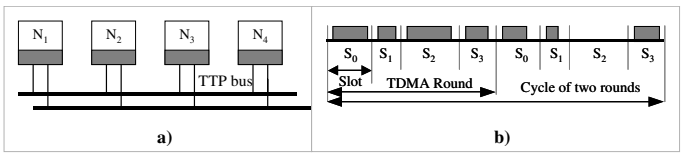


Figure 1. System Architecture Example

1. The number of faults *k* can be larger than the number of processors in the system.

2.2 Software Architecture and Fault-Tolerance Techniques

We have designed a software architecture which runs on the CPU in each node, and which has a real-time kernel as its main component. The processes are activated based on the local schedule tables, and messages are transmitted according to the MEDL. For more details about the software architecture and the message passing mechanism the reader is referred to [7]. The error detection and fault-tolerance mechanisms are part of the software architecture. The software architecture, including the real-time kernel, error detection and fault-tolerance mechanisms are themselves fault-tolerant. We use two mechanisms for tolerating faults: equidistant checkpointing with rollback recovery and active replication. Rollback recovery uses time redundancy to tolerate fault occurrences. Replication provides space redundancy that allows to distribute the timing overhead among several processors.

Let us consider the example in Figure 2, where we have process P_1 and a fault-scenario consisting of $k = 2$ transient faults that can happen during one cycle of operation.

- In Figure 2a we have the worst-case fault scenario for checkpointing, where we consider a single checkpoint for P_1. The worst-case *checkpointing overhead* (the time it takes, in the worst case, to save a checkpoint) for process P_1 is $\chi_1 = 5$ ms, and is depicted with a black rectangle. There are several strategies for saving the checkpoint data, such as memory or shared disc. Depending on the approach used, the designer will determine for each process P_i the worst-case checkpointing overhead χ_i. The first fault happens during the process P_1's execution, and is detected by the error detection mechanism. The worst-case *error detection overhead* for P_1 is $\alpha_1 = 5$ ms, and is depicted with a dark gray rectangle. After the error has been detected, the task is recovered based on the information in the saved checkpoint. After a worst-case *recovery overhead* of $\mu_1 = 5$ ms, depicted with a light gray rectangle, P_1 will be executed again. Its second execution in the worst-case could also experience a fault, and will be recovered based on the first checkpoint. Finally, the third execution of P_1 will take place without fault.
- In the case of active replication, depicted in Figure 2b, each replica is executed on a different processor. Three replicas are needed to tolerate the two possible faults and, in the worst-case scenario depicted in Figure 2b, only the execution of the replica on processor N_3 is successful. The error is detected by

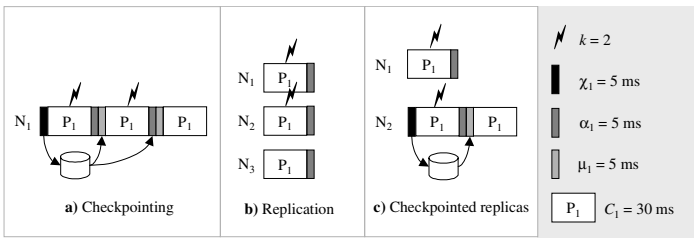


Figure 2. Worst-Case Fault Scenario and Fault-Tolerance Techniques

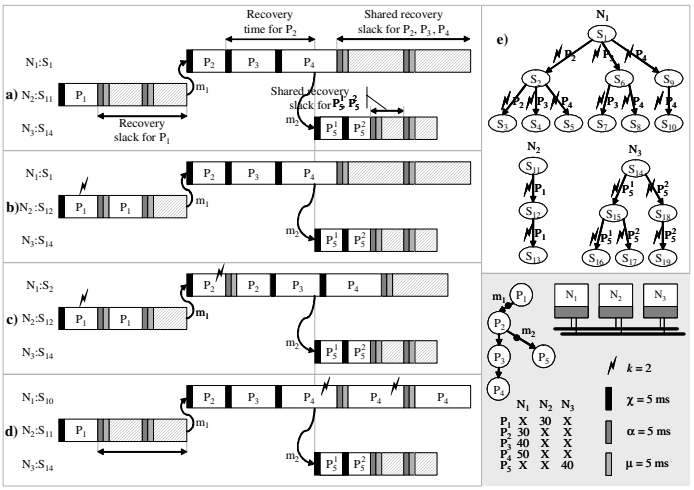


Figure 3. Transparent Checkpointing and Contingency Schedules

the error detection mechanism, and the worst-case error detection overhead is depicted as a dark grey rectangle in Figure 2b. With active replication, the input has to be distributed to all the replicas. Our execution model assumes that the descendants of replicas can start as soon as they have received the first valid message from a replica.

- In a third alternative, presented in Figure 2c, checkpointing and replication are combined for tolerating faults in a process. In this case, for tolerating the two faults we use two replicas and a checkpoint: process P_1 on node N_2, which is a replica of process P_1 on node N_1, is recovered using a checkpoint.

Let us consider the example[1] in Figure 3 where we have five processes, P_1 to P_5 mapped on three nodes, N_1 to N_3. The worst-case execution times for each process are given in the table below the application graph. An "X" in the table means that the particular process cannot be mapped on that node. We consider that at most two faults can occur, and the overheads of the fault-tolerance mechanisms (χ, α and μ) are given in the figure. All processes in the example use checkpointing: P_1 to P_4 have one checkpoint, while P_5 has two checkpoints. We denote with P_i^j that segment of P_i which follows after the j^{th} checkpoint of P_i. Thus, P_5^1 is the first segment of P_5, following the first checkpoint, while P_5^2 is the process segment following the second checkpoint (see Figure 3).

When checkpointing is used for tolerating faults, we have to introduce in the schedule table *recovery slack*, which is idle time on the processor needed to recover (re-execute) the failed process segment. For example, for P_1 on node N_2, we introduce a recovery slack of $k \times (C_1 + \alpha + \mu) = 80$ ms to make sure we can recover P_1 even in the case it experiences the maximum number of two faults (see Figure 3a). The recovery slack can be shared by several processes, like is the case of process P_2, P_3 and P_4 on node N_1, Figure 3a. This shared slack has to be large enough to accommodate the recovery of the largest process (in our case P_4) in the case of two faults. This slack can then handle two faults also in P_2 and P_3, which take less to execute than P_4. Note that the recovery slack for P_5, which has two checkpoints, is only half the size of P_5, since only one segment of P_5 (either P_5^1 or P_5^2) has to be recovered from its corresponding checkpoint, and not the whole process. In general, a process has an optimal number of checkpoints depending on its particular context in the schedule table, as discussed in Section 4.2.

Moreover, in this paper we use for checkpointing a particular type of recovery, called *transparent recovery* [14], that hides fault occurrences on a processor from other processors. On a processor N_i where a fault occurs, the scheduler has to switch to a *contingency schedule* that delays descendants of the faulty process running on the same processor N_i. However, a fault happening on another processor, is not visible on N_i, even if the descendants of the faulty process are mapped on N_i. For example, in Figure 3a, where we assume that no faults occur, in order to isolate node N_1 from the occurrence of a fault in P_1 on node N_2, message m_1 from P_1 to P_2 cannot be transmitted at the end of P_1's execution. Message m_1 has to arrive at the destination at a fixed time, regardless of what happens on node N_1, i.e., transparently. Consequently, m_1 can only be transmitted after a time $k \times (C_1 + \alpha + \mu)$, at the end of the recovery slack for P_1. Similarly, the transmission of m_2 also has to be delayed, to mask in the worst-case two consecutive faults in P_2. However, a fault in P_2 will delay processes P_3 and P_4 which are on the same processor (see in Figure 3c the time-line for node N_1).

Once a fault happens, the scheduler in a node has to switch to a *contingency schedule*. For example, once a fault occurs in process P_1 in the schedule depicted in Figure 3a, the scheduler on node N_2 will have to switch to the contingency schedule in Figure 3b, where P_1 is delayed with $C_1 + \alpha + \mu$ to account for the fault. A fault in P_2 will result in activating a contingency schedule on N_1 which contains a different start time not only for P_2, but also for P_3 and P_4 (see Figure 3c). There are several contingency schedules, depending on the combination of processes and faults. For two faults, and the processes in Figure 3, there are 19 contingency schedules, depicted in Figure 3e as a set of trees. There is one tree for each processor. One node S_i in the tree represents a contingency schedule, and the path from the root node of a tree to a node S_i, represents the sequence of faults (labelled with the process name in which the fault occurs) that lead to contingency schedule S_i. For example, in Figure 3a we assume that no faults occur, and thus we have the schedules S_1 on node N_1, S_{11} on N_2 and S_{14} on N_3. We denote such initial schedules with the term "*root schedule*", since they are the root of the contingency schedules tree. An error in P_1 (Figure 3b) will be observed by the scheduler on node N_2 which will switch to the contingency schedule S_{12}.

The end-to-end worst-case delay of the application is given by the maximum finishing time of any contingency schedule, since this is a situation that can happen in the worst-case scenario. For the application in Figure 3, the largest delay is produced by schedule S_{10}, which has to be activated when two consecutive faults happen in P_4 (Figure 3d). The end-to-end worst-case delay is equal to the size of the root schedules, including the recovery slack, depicted in Figure 3a. This is due to the fact that the root schedules have to contain enough recovery slack to accommodate even the worst-case scenario.

3. Application Model

We model an application $\mathcal{A}(\mathcal{V}, \mathcal{E})$ as a set of directed, acyclic, polar graphs $\mathcal{G}_i(\mathcal{V}_i, \mathcal{E}_i) \in \mathcal{A}$. Each node $P_i \in \mathcal{V}$ represents one pro-

1. Bus communication is ignored for this example, but is considered later.

cess. An edge $e_{ij} \in \mathcal{E}$ from P_i to P_j indicates that the output of P_i is the input of P_j. A process can be activated after all its inputs[1] have arrived and it issues its outputs when it terminates. The communication time between processes mapped on the same processor is considered to be part of the process worst-case execution time and is not modeled explicitly. Communication between processes mapped to different processors is performed by message passing over the bus. Such message passing is modeled as a communication process inserted on the arc connecting the sender and the receiver process.

The combination of fault-tolerance policies to be applied to each process is given by two functions.

- $\mathcal{F}$: $\mathcal{V} \rightarrow$ {*Replication*, *Checkpointing*} determines whether a process is replicated or checkpointed. When active replication is used for a process P_i, we introduce several replicas into the application $\mathcal{A}$, and connect them to the predecessors and successors of P_i. Let $\mathcal{V}_R$ be the set of replica processes introduced into the application.
- The second function $\mathcal{X}$: $\mathcal{V} \cup \mathcal{V}_R \rightarrow \aleph$ decides the number of checkpoints to be applied to a process in the application, including to the replicas in $\mathcal{V}_R$, if necessary, see Figure 2c. We consider equidistant checkpointing, thus the checkpoints are equally distributed throughout the execution time of the process. If for a process $P_i \in \mathcal{V} \cup \mathcal{V}_R$ we have $\mathcal{X}(P_i) = 0$, then it means that process P_i is not checkpointed.

Moreover, each process $P_i \in \mathcal{V}$ is characterized by an error detection overhead α_i and a recovery overhead μ_i. Each process P_i considered for checkpointing is also characterized by the checkpointing overhead χ_i.

The mapping of a process in the application is given by a function $\mathcal{M}$: $\mathcal{V} \cup \mathcal{V}_R \rightarrow \mathcal{N}$, where $\mathcal{N}$ is the set of nodes in the architecture. For a process $P_i \in \mathcal{V} \cup \mathcal{V}_R$, $\mathcal{M}(P_i)$ is the node to which P_i is assigned for execution. Each process can potentially be mapped on several nodes. Let $\mathcal{N}_{P_i} \subseteq \mathcal{N}$ be the set of nodes to which P_i can potentially be mapped. We consider that for each $N_k \in \mathcal{N}_{P_i}$, we know the worst-case execution time $C_{P_i}^{N_k}$ of process P_i, when executed on N_k. We also consider that the size of the messages is given.

4. Fault-Tolerant System Design

In this paper, by policy assignment we denote the decision whether a certain process should be checkpointed or replicated.

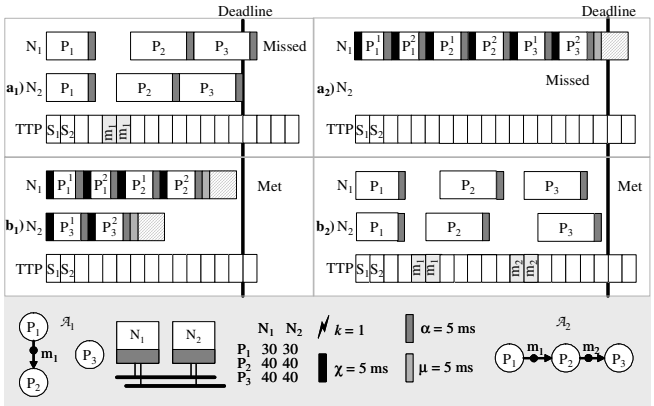


Figure 4. Comparison of Checkpointing and Replication

1. The first valid message from the replicas (identified by a "valid bit", part of the message format) is considered as the input.

Mapping a process means placing it on a particular node in the architecture.

There could be cases where the policy assignment decision is taken based on the experience and preferences of the designer, considering aspects like the functionality implemented by the process, the required level of reliability, hardness of the constraints, legacy constraints, etc. We denote with $\mathcal{P}_R$ the subset of processes which the designer has assigned replication, while $\mathcal{P}_C$ contains processes which are to be checkpointed.

Most processes, however, do not exhibit certain particular features or requirements which obviously lead to checkpointing or replication. Let $\mathcal{P}$ be the set of processes in the application $\mathcal{A}$. The subset $\mathcal{P}^+ = \mathcal{P} \setminus (\mathcal{P}_C \cup \mathcal{P}_R)$ of processes could use any of the two techniques for tolerating faults. Decisions concerning the policy assignment to this set of processes can lead to various trade-offs concerning, for example, the schedulability properties of the system, the amount of communication exchanged, the size of the schedule tables, etc.

For part of the processes in the application, the designer might have already decided their mapping. For example, certain processes, due to constraints like having to be close to sensors/actuators, have to be physically located in a particular hardware unit. They represent the set $\mathcal{P}_M$ of already mapped processes. Consequently, we denote with $\mathcal{P}^* = \mathcal{P} \setminus \mathcal{P}_M$ the processes for which the mapping has not yet been decided.

Our problem formulation is as follows:

- As an input we have an application $\mathcal{A}$ given as a set of process graphs (Section 3) and a system consisting of a set of nodes $\mathcal{N}$ connected using the TTP communication protocol.
- The fault model is given by the parameter k, which denotes the total number of transient faults that can appear in the system during one cycle of execution.
- As introduced previously, $\mathcal{P}_C$ and $\mathcal{P}_R$ are the sets of processes for which the fault-tolerance policy has already been decided. Also, $\mathcal{P}_M$ denotes the set of already mapped processes.

We are interested to find a system configuration ψ such that the k transient faults are tolerated and the imposed deadlines are guaranteed to be satisfied, within the constraints of the given architecture $\mathcal{N}$.

Determining a system configuration $\psi = <\mathcal{F}, \mathcal{X}, \mathcal{M}, \mathcal{S}>$ means:

1. finding a combination of fault-tolerance policies, given by $\mathcal{F}$ and $\mathcal{X}$, for each processes in $\mathcal{P}^+ = \mathcal{P} \setminus (\mathcal{P}_C \cup \mathcal{P}_R)$,
2. deciding on a mapping $\mathcal{M}$ for each process in $\mathcal{P}^* = \mathcal{P} \setminus \mathcal{P}_M$;
3. deciding on a mapping $\mathcal{M}$ for each replica in $\mathcal{V}_R$;
4. deriving the set $\mathcal{S}$ of root schedule tables on each processor and the MEDL for the TTP.

4.1 Fault-Tolerance Policy Assignment

Let us illustrate some of the issues related to policy assignment. In the example presented in Figure 4 we have the application $\mathcal{A}_1$ with three processes, P_1 to P_3, and an architecture with two nodes, N_1 and N_2. The worst-case execution times on each node are given in a table to the right of the architecture, and processes can be mapped to any node. The fault model assumes a single fault, thus $k = 1$, and the fault-tolerance overheads are presented in the figure. The application $\mathcal{A}_1$ has a deadline of 140 ms depicted with a thick vertical line. We have to decide which fault-tolerance technique to use.

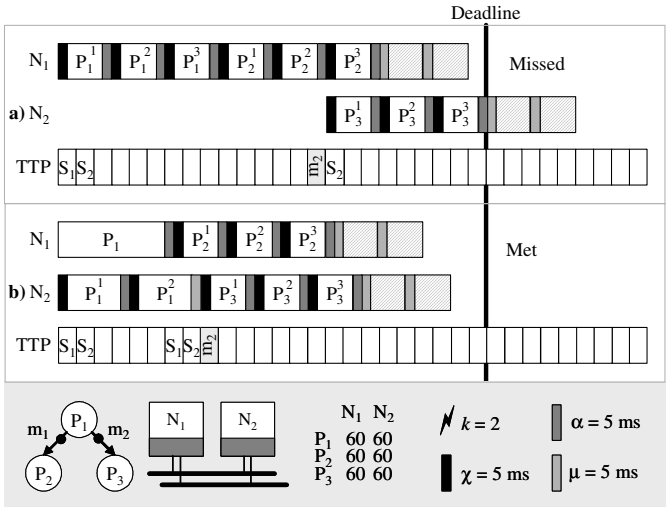


Figure 5. Combining Checkpointing and Replication

In Figure 4 we depict the root schedules[1] for each node, and for the TTP bus. Node N_1 is allowed to transmit in slot S_1, while node N_2 can use slot S_2. A TDMA round is formed of slot S_1 followed by slot S_2, each of 10 ms length. Comparing the schedules in Figure 4a$_1$ and 4b$_1$, we can observe that using active replication (a$_1$) the deadline is missed. However, using checkpointing (b$_1$) we are able to meet the deadline. Each process has an optimal number of two checkpoints in Figure 4b$_1$. If we consider application $\mathcal{A}_2$, similar to $\mathcal{A}_1$ but with process P_3 data dependent on P_2, the deadline of 180 ms is missed in Figure 4a$_2$ if checkpointing is used, and it is met when replication is used as in Figure 4b$_2$.

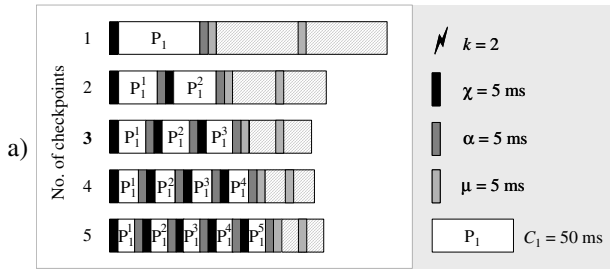
This example shows that the particular technique to use, has to be carefully adapted to the characteristics of the application. Moreover, the best result is most likely to be obtained when both techniques are used together, some processes being checkpointed, while others replicated.

Let us now consider the example in Figure 5, where we have an application with three processes, P_1 to P_3, mapped on an architecture of two nodes, N_1 and N_2. Node N_1 transmits in slot S_1, while node N_2 in S_2. Processes can be mapped to any node, and the worst-case execution times on each node are given in a table. In Figure 5a all processes are using checkpointing, and the depicted root schedule is optimal for this case. Note that m_2 can be transmitted only in the bus slot S_1 corresponding to node N_1, and has to be delayed to mask a potential failure of P_1 to node N_2. With this setting, using checkpointing will miss the deadline. However, combining checkpointing with replication, as in Figure 5b where process P_1 is replicated, will meet the deadline. In this case, message m_2 does not have to be delayed to mask the failure of process P_1. Instead, P_3 will receive m_2 from either P_1 or its replica.

4.2 Optimizing the Number of Checkpoints

Another optimization issue is related to the optimal number of checkpoints. Let us consider Figure 7a where we have process P_1 with a worst-case execution time of $C_1 = 50$ ms, and the fault scenario with $k = 2$ and all worst-case overheads equal to 5 ms. In Figure 7a we depict the execution time needed for P_1 to tolerate two faults, considering from one to five checkpoints. Since

1. The schedules depicted are optimal.



b) $$n_i^0 = \begin{cases} n_i^- = \left\lfloor \sqrt{\frac{kC_i}{\chi_i + \alpha_i}} \right\rfloor, \text{if } C_i \le n_i^-(n_i^- + 1)\frac{\chi_i + \alpha_i}{k} \\ n_i^+ = \left\lceil \sqrt{\frac{kC_i}{\chi_i + \alpha_i}} \right\rceil, \text{if } C_i > n_i^-(n_i^- + 1)\frac{\chi_i + \alpha_i}{k} \end{cases}$$

Figure 7. Locally Optimal Number of Checkpoints

P_1 has to tolerate two faults, the recovery slack has to be double of the size of a P_1 execution segment. Thus, for one checkpoint, the recovery slack is $(50 + 5) \times 2 = 110$ ms. If two checkpoints are introduced, in the case of an error only the segments P_1^1 or P_1^2 have to be recovered, not the whole process, thus the recovery slack is reduced to 60 ms. By introducing more checkpoints, the recovery slack can be thus reduced. However, there is a point over which the reduction in the recovery slack is offset by the increase in the overhead related to setting each checkpoint. For process P_1 in Figure 7a, going beyond three checkpoints will enlarge the total execution time. Let n_i^0 be the optimal number of checkpoints for P_i, when P_i is considered in isolation. Punnekkat et al. [23] derive a formula for n_i^0 in the context of preemptive scheduling, single fault assumption. We have updated this formula to consider that several faults can occur, and to consider our detailed checkpointing overheads (see Figure 7b). For the example, in Figure 7a, $n_1^0 = 3$.

The equation in Figure 7b allows us to calculate the optimal number of checkpoints for a certain process *considered in isolation*. However, calculating the number of checkpoints for each individual process will not produce a solution which is optimal for the whole application.

Let us consider the example in Figure 6, where we have three processes, P_1 to P_3 on a single processor system. We consider two transient faults, and the process worst-case execution times and the fault-tolerance overheads are depicted in the figure. The locally optimal checkpoints for P_1, P_2 ad P_3 are $n_1^0 = 1$, $n_2^0 = 3$ and $n_3^0 = 3$, respectively. Using these checkpoints will lead to a deadline miss, as shown in Figure 6a. However, the globally op-

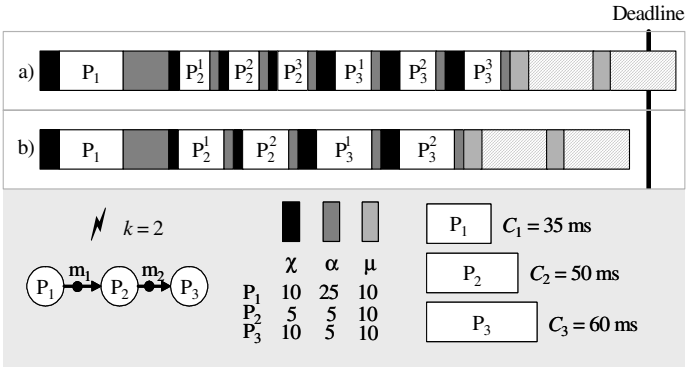


Figure 6. Optimizing the Number of Checkpoints

timal number of checkpoints is $n_1^0 = 1$, $n_2^0 = 2$ and $n_3^0 = 2$ (i.e., P_2 and P_3 should have two checkpoints instead of three). In this case, presented in Figure 6b, the deadline will be met.

5. Design Optimization Strategy

The design problem formulated in the previous section is NP-complete (both the scheduling and the mapping problems, considered separately, are already NP-complete [10]). Therefore, our strategy is to elaborate a heuristic and divide the problem into several, more manageable, subproblems. Our optimization strategy which produces the configuration ψ leading to a schedulable fault-tolerant application is outlined in Figure 8 and has three steps:

1. In the first step (lines 1–3) we quickly decide on an initial bus access configuration $\mathcal{B}^0$, an initial fault-tolerance policy assignment given by $\mathcal{F}^0$ and $\mathcal{X}^0$, and an initial mapping $\mathcal{M}^0$. The initial bus access configuration (line 1) is determined by assigning nodes to the slots ($S_i = N_i$) and fixing the slot length to the minimal allowed value, which is equal to the length of the largest message in the application. The initial mapping and fault-tolerance policy assignment algorithm (InitialMPA line 2 in Figure 8) assigns a checkpointing policy with a locally optimal number of checkpoints (using the equation in Figure 7b) to each process in $\mathcal{P}^+$ and produces a mapping for the processes in $\mathcal{P}^*$ that tries to balance the utilization among nodes and buses. The application is then scheduled using the ListScheduling algorithm (see Section 5.1). If the application is schedulable the optimization strategy stops.
2. If the application is not schedulable, we use, in the second step, a tabu search-based algorithm discussed in Section 5.2.
3. Finally, in step three, the bus access optimization is performed [7].

If after these two steps the application is unschedulable, we conclude that no satisfactory implementation could be found with the available amount of resources.

5.1 Static Scheduling

Once a fault-tolerance policy and a mapping are decided, as well as a communication configuration is fixed, the processes and messages have to be scheduled. We use a static scheduling algorithm for building the schedule tables for the processes and deriving the MEDL for messages.

We have adapted the list-scheduling based algorithm from [13], where we used re-execution and replication for tolerating faults. The scheduling algorithm is responsible for deriving offline the root schedules. The contingency schedules are determined online, by the scheduler in each node, starting from a current schedule, based on the occurrences of faults, is able to derive *online*, in linear time, the necessary contingency schedule. Given a current schedule S_i on a node N_j, and an error detected in a process P_k, the scheduler on node N_j traverses S_i and increments the start time of P_k and of the following processes running on the same node. The start time of P_k is incremented with $C_k + \alpha + \mu$. The start time of a process P_l following P_k in the schedule table S_i will be incremented with $C_k + \alpha + \mu - slack(P_k, P_l)$, where $slack(P_k, P_l)$ is the idle time (including idle time, e.g. waiting for an incoming message, and recovery slack) in S_i between P_k and P_l. The overhead due to obtaining the new start times is considered in the schedule table construction.

```
OptimizationStrategy(A, N)
1    Step 1:   B^0 = InitialBusAccess(A, N)
2              <M^0, F^0, X^0> = InitialMPA(A, N, B^0)
3              S^0 = ListScheduling(A, N, M^0, F^0, X^0)
4              if S^0 is schedulable then return ψ^0 end if
5    Step 2:   ψ = TabuSearchMPA(A, N, ψ^0)
6              if S is schedulable then return ψ end if
7    Step 3:   B = BusAccessOptimization(A, N, ψ)
8              S = ListScheduling(A, N, M, F, X)
9    return ψ
end OptimizationStrategy
```

Figure 8. The General Optimization Strategy

5.2 Mapping and Fault-Policy Assignment

For deciding the mapping and fault-policy assignment we use a tabu search based heuristic approach, TabuSearchMPA. TabuSearchMPA uses design transformations (moves) to change a design such that the end-to-end delay of the root schedule is reduced. In order to generate neighboring solutions, we perform the following types of transformations:

- changing the mapping of a process;
- changing the combination of fault-tolerance policies for a process;
- changing the number of checkpoints used for a process.

The tabu search takes as an input the merged application graph Γ (obtained by merging the application graphs as detailed in [22], with a period equal to the LCM of all constituent graphs), the architecture $\mathcal{N}$ and the current implementation ψ, and produces a schedulable and fault-tolerant implementation x^{best}. The tabu search is based on a neighborhood search technique, and thus in each iteration it generates the set of moves N^{now} that can be performed from the current solution x^{now}. The cost function to be minimized by the tabu search is the end-to-end delay of the root schedule produced by the list scheduling algorithm. In order to reduce the huge design space, in our implementation, we only consider changing the mapping or fault-tolerance policy of the processes on the critical path. We define the critical path as the path through the merged graph Γ which corresponds to the longest delay in the schedule table.

Moreover, we also try to eliminate moves that change the number of checkpoints if it is clear that they do not lead to better results. Consider the example in Figure 9 where we have four processes, P_1 to P_4 mapped on two nodes, N_1 and N_2. The worst-case execution times of processes and their fault-tolerance overheads are also given in the figure, and we can have at most two faults. The number of checkpoints calculated using the formula in Figure 7b are: $n_1^0 = 2$, $n_2^0 = 2$, $n_3^0 = 1$ and $n_4^0 = 3$. Let us assume that our current solution is the one depicted in Figure 9a, where we have $\mathcal{X}(P_1) = 2$, $\mathcal{X}(P_2) = 1$, $\mathcal{X}(P_3) = 1$ and $\mathcal{X}(P_4) = 2$. Given a process P_i, with a current number of checkpoints $\mathcal{X}(P_i)$, our tabu search approach will generate moves with all possible checkpoints starting from 1, up to n_i^0. Thus, starting from the solution Figure 9a, we can have the following moves that modify the number of checkpoints: (1) decrease the number of checkpoints for P_1 to 1; (2) increase the number of checkpoints for P_2 to 2; (3) increase the number of checkpoints for P_4 to 3; (4) decrease the number of checkpoints for P_4 to 1. Moves (1) and (3) will lead to the optimal checkpoints depicted in Figure 9b.

In order to reduce optimization time, our heuristic will not try moves (2) and (4), since they cannot lead to a shorter critical

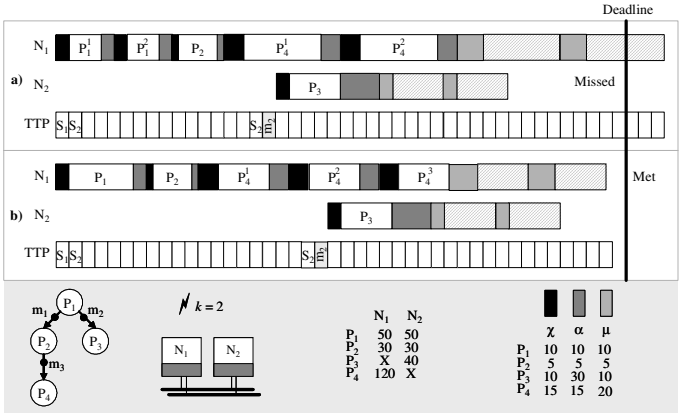


Figure 9. Restricting the Moves for Setting the Number of Checkpoints

path, and, thus, a better root schedule. Regarding move (2), by increasing the number of checkpoints for P_2 we can reduce its recovery slack. However, P_2 shares its recovery slack with P_1 and segments of P_4, which have a larger execution time, and thus even if the necessary recovery slack for P_2 is reduced, it will not affect the size of the shared slack (and implicitly, of the root schedule) which is given by the largest process (or process segment) that shares the slack. Regarding move (4), we notice that by decreasing for P_4 the number of checkpoints to 1, we increase the recovery slack, which, in turn, increases the length of the root schedule.

6. Experimental Results

For the evaluation of our algorithms we used applications of 20, 40, 60, 80, and 100 processes (all unmapped and with no fault-tolerance policy assigned) implemented on architectures consisting of 3, 4, 5, 6, and 7 nodes, respectively. We have varied the number of faults depending on the architecture size, considering 4, 5, 6, 7, and 8 faults for each architecture dimension, respectively. We have also varied the fault-tolerance overheads for each process, from 1% of its worst-case execution time up to 30%. Fifteen examples were randomly generated for each application dimension, thus a total of 75 applications were used for experimental evaluation. The experiments were performed on Sun Fire V250 computers.

We were first interested to evaluate the proposed optimization strategy in terms of overheads introduced due to fault-tolerance. For this, we have implemented each application without any fault-tolerance concerns. This non-fault-tolerant implementation, NFT, has been obtained using an approach similar to the algorithm in Figure 8 but without fault-tolerance techniques. The same applications have been implemented on the same amount of resources, using the optimization strategy in Figure 8, with multiple checkpoints and replication (MCR). Together with the MCR approach we have also evaluated two extreme approaches: MC that considers only checkpointing, and MR which relies only on replication for tolerating faults. MC and MR use the same optimization approach as MCR, but besides the mapping moves, they consider assigning only checkpointing (including the optimization of the number of checkpoints) or only replication, respectively. In addition, we have also implemented a checkpointing-only strategy, namely MC0, similar to MC, but where the number of checkpoints are fixed based on the formula in Figure 7b, updated from [23]. For these experiments, we have derived the shortest schedule within an imposed time limit for optimization: 1 minute for 20 processes, 10 for 40, 30 for 60, 2 hours and 30 min. for 80 and 6 hours for 100 processes.

Table 1. Fault-Tolerance Overheads

Number of processes	Maximum overhead	Average overhead	Minimum Overhead
20	150.00%	102.59%	74.90%
40	130.23%	111.66%	99.43%
60	110.62%	90.85%	75.41%
80	91.52%	73.08%	59.83%
100	73.50%	64.27%	56.39%

Let δ_{MCR} and δ_{NFT} be the lengths of the root schedules obtained using MCR and NFT, respectively. The overhead is defined as $100 \times (\delta_{MCR} - \delta_{NFT}) / \delta_{NFT}$. The fault-tolerance overheads of MCR compared to NFT are presented in Table 1. The MCR approach can offer fault-tolerance within the constraints of the architecture at an average overhead of 88.49%. In the case only replication is used (MR), the overheads compared to NFT are very large (e.g., 306.51% on average for applications 100 processes).

We were interested to compare the quality of MCR to MC0, MC and MR. In Figure 10a-c we show the average percentage deviation of overheads obtained with MCR and MC from the baseline represented by MC0 (larger deviation means smaller overhead). From Figure 10 we can see that by optimizing the combination of checkpointing and replication MCR performs much better compared to MC and MC0. This shows that considering checkpointing at the same time with replication can lead to significant improvements. Moreover, by considering the global optimization of the number of checkpoints, with MC, improvements can be gained over MC0 which computes the optimal number of checkpoints for each process in isolation.

In Figure 10a we consider 4 processors, 3 faults, and vary the application size from 40 to 100 processes. As the amount of available resources per application decreases, the improvement due to replication (part of MCR) will diminish, leading to a result comparable to MC.

In Figure 10b, we were interested to evaluate our MCR approach in the case the checkpointing overheads (i.e., $\chi + \alpha$, see Figure 2) are varied. We have considered applications with 40 processes mapped on four processors, and we have varied the checkpointing overheads from 2% of the worst-case execution time of a process up to 60%. We can see that as the amount of checkpointing overheads increases, our optimization approaches are able to find increasingly better quality solutions compared to MC0.

We have also evaluated the MCR and MC approaches in the case the number of transient faults increase. We have considered applications with 40 processes mapped on 4 processors, and varied k from 2 to 6, see Figure 10c. As the number of faults increase, it is more difficult for our optimization heuristics to improve compared to MC0, and the improvement achieved over MC0 will stabilize to about 10% improvement (e.g., for $k = 10$, not shown in the figure, the improvement due to MC is 8.30%, while MCR improves with 10.29%).

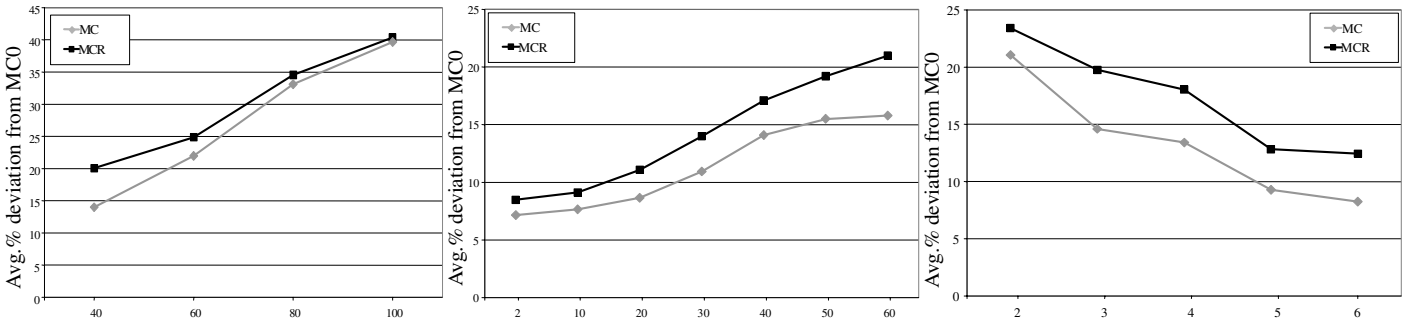


Figure 10. MCR and MC compared to MC0

Finally, we considered a real-life example implementing a vehicle cruise controller (CC). The process graph that models the CC has 32 processes, and is described in [22]. The CC was mapped on an architecture consisting of three nodes: Electronic Throttle Module (ETM), Anti-lock Breaking System (ABS) and Transmission Control Module (TCM). We have considered a deadline of 260 ms, $k = 2$ faults and the checkpointing overheads are 10% of the worst-case execution time of the processes.

In this setting, the MCR produced a schedulable fault-tolerant implementation with a worst-case system delay of 230 ms, and with an overhead compared to NFT (which produces a non-fault-tolerant schedule of length 136 ms) of 69%. If we globally optimize the number of checkpoints using MC we obtain a schedulable implementation with a delay of 256 ms, compared to 276 ms produced by MC0 which is larger than the deadline. If replication only is used, as in the case of MR, the delay is 320 ms, which is greater than the deadline.

7. Conclusions

In this paper we have addressed the synthesis of distributed embedded systems for fault-tolerant hard real-time applications. The processes are scheduled with static cyclic scheduling, while for the message transmission we use the TTP. We have extended our previous work to handle checkpointing, which provides time-redundancy, at the same time with replication, which provides space-redundancy. We have shown that by globally optimizing the number of checkpoints significant improvements can be gained.

We have proposed a synthesis algorithm based on tabu-search, which derives schedulable fault-tolerant implementations. The proposed approach has been evaluated using extensive experimental data, including a real-life example.

References

[1] A. Bertossi, L. Mancini, "Scheduling Algorithms for Fault-Tolerance in Hard-Real Time Systems", *Real Time Systems*, 7(3), 229–256, 1994.

[2] A. Burns, R. I. Davis, S. Punnekkat, "Feasibility Analysis for Fault-Tolerant Real-Time Task Sets", *Proc. of Euromicro Workshop on R-T Systems*, 29–33, 1996.

[3] P. Chevochot, I. Puaut, "Scheduling Fault-Tolerant Distributed Hard-Real Time Tasks Independently of the Replication Strategies", *Proc. of the Real-Time Computing Systems and Applications Conference*, 356–363, 1999.

[4] V. Claeson, S. Poldena, J. Söderberg, "The XBW Model for Dependable Real-Time Systems", *Proc. Parallel and Distributed Systems Conf.*, 1998.

[5] C. Dima et al, "Off-line Real-Time Fault-Tolerant Scheduling", *Proc.of the Euromicro Parallel and Distributed Processing Workshop*, 410–417, 2001.

[6] E. N. Elnozahy et al., "A survey of rollback-recovery protocols in message-passing systems", *ACM Computing Surveys*, 34(3), 375–408, 2002.

[7] P. Eles, A. Doboli, P. Pop, Z. Peng, "Scheduling with Bus Access Optimization for Distributed Embedded Systems", *IEEE Transactions on VLSI Systems,* 8(5), 472–491, 2000.

[8] G. Fohler, "Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems", *Proceedings of the Real-Time Systems Symposium*, 152–161, 1995.

[9] G. Fohler, "Adaptive Fault-Tolerance with Statically Scheduled Real-Time Systems", *Proc. Euromicro Real-Time Systems Workshop*, 161–167, 1997.

[10] M. R. Garey, D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 2003.

[11] C. C. Han, K. G. Shin, J. Wu, "A Fault-Tolerant Scheduling Algorithm for Real-Time Periodic Tasks with Possible Software Faults", *IEEE Transactions on Computers*, 52(3), 362–372, 2003.

[12] K. Hoyme, K. Driscoll, "SAFEbus", *IEEE Aerospace and Electronic Systems Magazine*, 8(3), 34–39, 1992.

[13] V. Izosimov, P. Pop, P. Eles, Z. Peng, "Design Optimization of Time- and Cost-Constrained Fault-Tolerant Distributed Embedded Systems", *Proc. Design Automation and Test in Europe Conference*, 864-869, 2005.

[14] N. Kandasamy, J. P. Hayes, B. T. Murray, "Transparent Recovery from Intermittent Faults in Time-Triggered Distributed Systems", *IEEE Transactions on Computers*, 52(2), 113–125, 2003.

[15] N. Kandasamy, J. P. Hayes, B.T. Murray "Dependable Communication Synthesis for Distributed Embedded Systems," *Proceedings of the Computer Safety, Reliability and Security Conference*, 275–288, 2003.

[16] H. Kopetz, *Real-Time Systems–Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, 1997.

[17] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, R. Zainlinger, "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach", *IEEE Micro*, 9(1), 25–40, 1989.

[18] H. Kopetz, G. Bauer, "The Time-Triggered Architecture", *Proceedings of the IEEE*, 91(1), 112–126, 2003.

[19] A. Orailoglu, R. Karri, "Coactive scheduling and checkpoint determination during high level synthesis of self-recovering microarchitectures", *IEEE Transactions on VLSI Systems*, 2/3, 304–311, September 1994.

[20] C. Pinello, L. P. Carloni, A. L. Sangiovanni-Vincentelli, "Fault-Tolerant Deployment of Embedded Software for Cost-Sensitive Real-Time Feedback-Control Applications", *Proceedings of the Design Automation and Test in Europe Conference*, 1164–1169, 2004.

[21] S. Poldena, *Fault Tolerant Systems—The Problem of Replica Determinism*, Kluwer Academic Publishers, 1996.

[22] P. Pop, P. Eles, Z. Peng, *Analysis and Synthesis of Distributed Real-Time Embedded Systems*, Kluwer Academic Publishers, 2004.

[23] S. Punnekkat, A. Burns, "Analysis of Checkpointing for Schedulability of Real-time Systems", *Proc. Real-Time Systems Symposium*, 198–205, 1997.